Étape 11 : Déplacement sensoriel, partie 2.

Jamila Sam & Jean-Cédric Chappelier, 2018

Version : 1.0

But

Mettre en place les stratégies spécifiques de déplacement pour les animaux et finaliser l'intégration des phéromones

Description

Nous allons dans cette partie adapter le code existant pour être capable d'y ajouter le déplacement sensoriel.

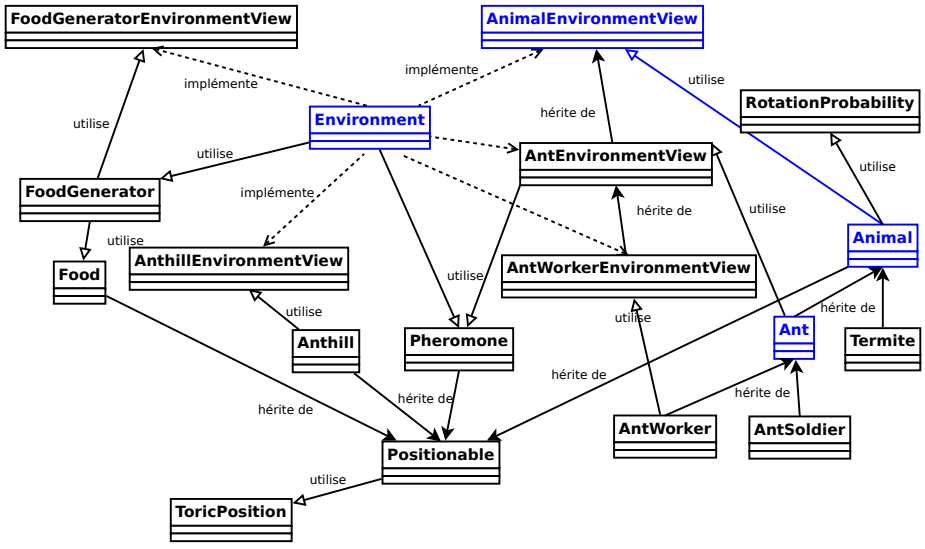

Figure 1: Composants impliqués dans l'étape 11 (en noir les composants déjà codés, en vert les nouveaux composants, en bleu ceux retouchés)

Dans les étapes précédentes, la seule stratégie de rotation (changement de direction) possible pour les animaux en cours de déplacement se résumait à des petits changements aléatoires des angles de direction, indépendants de l'environnement et de son contenu.

Le but est de faire en sorte que pour les fourmis, il puisse y avoir une stratégie différente, influencée par la présence de phéromone. Plus généralement, nous souhaitons garantir que n'importe quel nouveau type d'animal puisse bénéficier d'une stratégie de rotation spécifique et potentiellement dépendante de l'environnement.

Cela va induire une conception très proche de ce que nous avons mis en place pour implémenter les *comportements spécifiques des animaux*; sauf que cette fois il s'agira d'implémenter leurs *stratégies de rotation spécifiques*. Nous aurons donc à nouveau recours au schéma du « *double dispatch* » qui impliquera de retoucher aux composants `Environment`, `AnimalEnvironmentView`, `Animal` et `Ant`, comme le montre la figure 1.

Nous recourrons une troisième fois au « *double dispatch* » pour implémenter les *actions spécifiques* qu'entreprennent les animaux *après un pas de déplacement* (comme le fait de lâcher de la phéromone pour les fourmis). Ceci impliquera de faire des retouches analogues à celles liées aux stratégies de rotations spécifiques.

Implémentation

Modification de `move`

La méthode `move()` de `Animal` doit donc désormais dépendre aussi de l'environnement (par exemple, pour que le déplacement des fourmis puisse dépendre des quantités de phéromone disponibles).

Nous allons pour cela réadapter la méthode `move()` pour qu'elle prenne en paramètre une vue sur l'environnement. Sa signature devient donc :

```
void move(AnimalEnvironmentView env, Time dt)
```

Adaptation de `computeRotationProbs()`

La méthode de déplacement pour les fourmis reste globalement la même que pour les animaux. Ce qui va changer essentiellement est la stratégie de rotation et donc le calcul des probabilités de déplacement associées aux intervalles d'angles entourant la fourmi.

Concrètement, la méthode `computeRotationProbs()` doit être spécialisée pour les fourmis, quelque soit leur type, de sorte à permettre le déplacement sensoriel en tenant compte de la présence de phéromones.

Or, pour le moment, la méthode `computeRotationProbs()` de la classe `Animal` ne prend pas de paramètre ; les probabilités ne dépendent donc pas de l'environnement. Les fourmis ayant besoin d'avoir accès aux phéromones pour le calcul des probabilités, il faudrait donc modifier `computeRotationProbs()` pour qu'elle prenne en paramètre une vue sur l'environnement. Cependant, cette méthode est appelée par `move()` et cette dernière ne peut donc lui passer en paramètre qu'un `AnimalEnvironmentView`; ce qui est trop générique pour ce qui nous intéresse (nous voulons que les animaux ne puissent pas tous systématiquement utiliser de la phéromone pour se déplacer).

Vous l'aurez reconnu, nous sommes ici face à une problématique tout à fait analogue à ce que nous avons expérimenté dans l'étape 8 et que nous avons pu résoudre par le biais du « *double dispatch* ». C'est l'environnement qui doit décider quel traitement il autorise pour tel ou tel animal relativement aux stratégies de rotation (et utilisation de la phéromone).

Nous allons donc apporter les modifications suivantes dans `Animal` :

- renommer la méthode `computeRotationProbs()` en `computeDefaultRotationProbs()` ; si une classe dérivant d'`Animal` peut se contenter d'un comportement par défaut dans le calcul des probabilités, elle pourra alors utiliser cette méthode ; Cette méthode sera déclarée comme `final` car il n'y a pas beaucoup de sens à redéfinir cette stratégie par défaut ;
- ajouter l'équivalent de `specificBehaviorDispatch` (mais cette fois pour la stratégie de rotation), à savoir une nouvelle méthode *abstraite*

```
RotationProbability computeRotationProbsDispatch(AnimalEnvironmentView env)
```

Dans `Ant` nous allons introduire la méthode `final`

```
RotationProbability computeRotationProbs(AntEnvironmentView env)
```

qui s'occupera de calculer les probabilités associées aux angles de direction en tenant compte de la vue qu'une fourmi a sur l'environnement (et qui lui permettre d'accéder aux phéromones). Une première implémentation de cette méthode est suggérée un peu plus bas.

Pour compléter le schéma du double dispatch nous allons :

- dans `Ant` implémenter `computeRotationProbsDispatch()` (voir plus bas); cette méthode est l'équivalent des méthodes `specificBehaviorDispatch` précédemment introduites pour les `AntWorker`et `AntSoldier` ; nous pouvons cependant déjà la coder à ce niveau de la hiérarchie car toutes les fourmis auront la même stratégie de rotation ;
- dans `AnimalEnvironmentView` la méthode

```
RotationProbability selectComputeRotationProbsDispatch(Ant ant)
```

qui est l'équivalent des méthodes `selectSpecificBehaviorDispatch` déjà introduites dans cette interface;

L'idée est la suivante :

- la méthode `move()` n'utilisera plus `computeRotationProbs()`, mais `computeRotationProbsDispatch()` (méthode qui permettra à l'environnement de sélectionner la stratégie de rotation permise pour l'animal) ;
- la classe `Ant` implémentera `computeRotationProbsDispatch()` ; elle devra simplement appeler `selectComputeRotationProbsDispatch()` de `AnimalEnvironmentView` ; cette méthode devra être la même pour chaque type de fourmi ; elle ne pourra donc pas être redéfinissable par les sous-classes ;
- la définition de `selectComputeRotationProbsDispatch()` dans `Environment` appellera enfin la méthode `computeRotationProbs()` de `Ant` ;

Le calcul des probabilités de rotation en fonction de la présence de phéromone est relativement complexe et fera l'objet de l'étape suivante du projet. Pour le moment, `computeRotationProbs()` se contentera donc retourner les probabilités de rotation par défaut (qui peuvent être obtenues par `computeDefaultRotationProbs()`).

Comportement supplémentaire lors d'un déplacement

La méthode `move()` étant `final`, il n'est pas possible pour les sous-classes de la redéfinir. En effet, nous souhaitons que tout animal possède le même modèle de déplacement. Toutefois, il se pourrait que pour une classe héritant d'`Animal` un comportement supplémentaire doive systématiquement être présent lors de l'appel à `move()`. Dans notre cas, nous aimerions qu'à chaque déplacement de la fourmi, celle-ci disperse de la phéromone. Une solution simple et correcte consiste à ajouter ce comportement après chaque appel à `move()` (au moyen d'un appel de méthode par exemple). Cependant, si la classe en question souhaitait « imposer » ce comportement à ses propres sous-classes, cette approche n'est pas viable. En effet, les sous-classes ne sont pas « obligées » d'appeler la méthode mettant en œuvre ce comportement supplémentaire après chaque appel à `move()`. C'est notre cas : nous souhaiterions que toute sous-classe de `Ant` appelant `move()` appelle également `spreadPheromones`.

Une solution possible consiste à ajouter une méthode abstraite `afterMove` et qui serait appelée *dans* `move()`. Cette nouvelle méthode implémenterait alors le comportement supplémentaire souhaité. Ainsi, à chaque appel de `move()`, cette méthode serait « automatiquement » appelée. De plus, les classes qui souhaiteraient imposer un comportement après chaque appel de `move()` peuvent déclarer `afterMove` comme `final` ; ainsi, il ne serait pas possible aux sous-classes d'échapper aux règles imposées par leur super-classe.

Cependant, la méthode `afterMove` devrait prendre les mêmes paramètres que `move()`, c.-à-d. `AnimalEnvironmentView` et `Time`. Vous l'aurez sans doute compris, qui dit `AnimalEnvironmentView` dit double dispatch... Et cette méthode ne fait pas exception à la règle !

La méthode `move()` appellera donc en guise et place de `afterMove`, une méthode abstraite (protégée)

```
void afterMoveDispatch(AnimalEnvironmentView env, Time dt)
```

une fois le déplacement effectué.

Pour le reste, le schéma reste exactement le même que pour les autres cas :

- les sous-classes d'`Animal` devront implémenter `afterMoveDispatch()` ; ici, seule `Ant` implémentera cette méthode (`AntSoldier` et `AntWorker` hériteront de cette implémentation) ;
- celle-ci devra simplement appeler la méthode

  ```
  void selectAfterMoveDispatch(Ant ant, Time dt)
  ```

de `AnimalEnvironmentView` que vous devrez également écrire ; son implémentation est simple : elle se contente d'appeler

```
void afterMoveAnt(AntEnvironmentView env, Time dt)
```

`(voir ci-dessous)` ;

- la méthode

  ```
  void afterMoveAnt(AntEnvironmentView env, Time dt)
  ```

`de `Ant` appellera simplement `spreadPheromones()` ; notez qu'on aurait également pu appeler`

Veillez également à déclarer `final` les nouvelles méthodes de `Ant` pour empêcher leur redéfinition dans ses sous-classes (sinon tout ce travail ne servirait à rien...).

Enfin, vu que les fourmis sont désormais capables de lâcher des traces de phéromone, il devient nécessaire d'intégrer le dessin de ces dernières à la méthode de rendu de `Environment`. Vous ajouterez donc à la méthode `Environment.renderEntities` le code nécessaire.

Tests et soumission

Tests locaux

Nous vous encourageons, comme toujours à d'un côté continuer d'ajouter vos propres tests ponctuels dans le fichier `ch/epfl/moocprog/tests/Main.java` et d'un autre côté utiliser l'interface graphique pour tester globalement le comportement de tout le système.

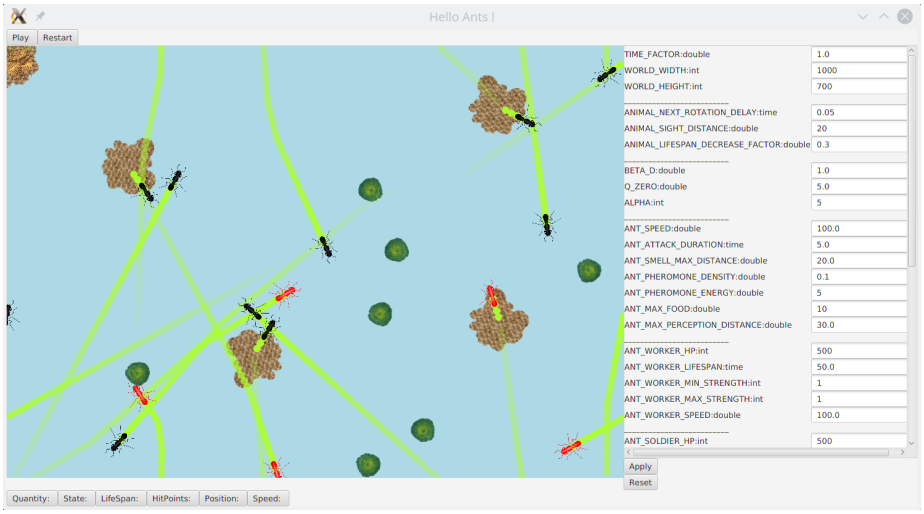

Figure 2: Exemple d'exécution à la fin de l'étape 11

L'application graphique devrait vous permettre d'observer :

- que les fourmis continuent à se déplacer selon la stratégie du déplacement inertiel comme elle l'ont fait jusqu'ici (vérifier que changer la fréquence des rotation a bien l'impact voulu sur les changements de direction aléatoires) ;
- que les fourmis déposent des traces de phéromone en se déplaçant (chemins en vert) ;
- que la phéromone s'évapore au cours du temps (devient plus claire jusqu'à disparaître) ;
- que la phéromone n'a pas d'incidence sur le déplacement (les fourmis peuvent croiser des chemins de phéromone sans y être attirées).

Vous pouvez profiter du rendu graphique pour vérifier des éléments qu'il était difficile de tester lors des étapes précédentes. Vérifiez par exemple que le taux d'évaporation des phéromones a bien une incidence (à zéro il n'y a plus d'évaporation).

La figure 2 donne un exemple d'exécution où l'on peut voir l'affichage des phéromones.

Une petite vidéo d'exemple d'exécution est disponible dans la matériel de la semaine 6.

Les fourmis ne sont pas encore sensibles à la présence de la phéromone, c'est le but de la prochaine étape d'y remédier.

Soumission

Pour soumettre votre devoir au correcteur automatique, il faut créer un fichier ZIP contenant **tous** vos fichiers sources personnels, depuis la racine `ch/` ; ceux de cette étape, mais aussi ceux de l'étape précédente. Concrètement, pour ce devoir ci, votre fichier ZIP doit donc contenir exactement les fichiers suivants :

```
ch/epfl/moocprog/AnimalEnvironmentView.java
ch/epfl/moocprog/Animal.java
ch/epfl/moocprog/AntEnvironmentView.java
ch/epfl/moocprog/AnthillEnvironmentView.java
ch/epfl/moocprog/Anthill.java
ch/epfl/moocprog/Ant.java
ch/epfl/moocprog/AntSoldier.java
ch/epfl/moocprog/AntWorkerEnvironmentView.java
ch/epfl/moocprog/AntWorker.java
ch/epfl/moocprog/Environment.java
ch/epfl/moocprog/FoodGeneratorEnvironmentView.java
ch/epfl/moocprog/FoodGenerator.java
ch/epfl/moocprog/Food.java
ch/epfl/moocprog/Pheromone.java
ch/epfl/moocprog/Positionable.java
ch/epfl/moocprog/RotationProbability.java
ch/epfl/moocprog/Termite.java
ch/epfl/moocprog/ToricPosition.java
```

Note : si c'est plus pratique pour vous, vous *pouvez* aussi faire un zip qui contient tout le sous-répertoire `ch/epfl/moocprog`, y compris, donc, les fichiers que nous vous avons fournis. Ces fichiers seront simplement ignorés par le correcteur automatique (et remplacés par les nôtres). Ces fichiers seront simplement ignorés par le correcteur automatique (et remplacés par les nôtres). L'énoncé « *Procédure de soumission* » de la semaine 1 vous indique comment procéder. **Avant de passer à l'étape suivante, n'oubliez pas de faire une sauvegarde de l'étape en cours**, comme indiqué dans le même document.